# Increasing the Power Efficiency of Bloom Filters for Network String Matching

Ilhan Kaya and Taskin Kocak
School of Electrical Engineering and Computer Science
University of Central Florida
Orlando, FL 32816
e-mail: {ikaya, tkocak}@cs.ucf.edu

***Abstract*— Although software based techniques are widely accepted in computer security systems, there is a growing interest to utilize hardware opportunities in order to compensate for the network bandwidth increases. Recently, hardware based virus protection systems have started to emerge. These type of hardware systems work by identifying the malicious content and removing it from the network streams. In principle, they make use of string matching. Bit by bit, they compare the virus signatures with the bit strings in the network. The Bloom filters are ideal data structures for string matching. Nonetheless, they consume large power when many of them used in parallel to match different virus signatures. In this paper, we propose a new type of Bloom filter architecture which exploits well-known pipelining technique.**

## I. Introduction

Applications providing intrusion detection, virus prevention, and content filtering have not kept pace with the increase in network speeds since they lack hardware functionality supporting them. There is a need to scan entire network packets bit by bit to determine predefined signatures for viruses and worms.

Before the packets are processed by the upper OSI layers, malicious packets have to be filtered. Bloom filters [2] are used in such filtering applications to match strings such as Snort rules [10]. Moreover, Bloom filters have been used for many network applications like web cache sharing [6], resource routing [5], string matching [1], [7]. A hardware system, consisting of Bloom filters to detect malignant content, is described in [7]. A detailed survey of Bloom filters for networking applications can be found in [3].

Although Bloom filters have found wide spread usage in networking applications, they are not conservative in terms of power. A network intrusion detection system (NIDS) consists of 4 Bloom filter engines can dissipate up to 5 W. In order to decrease the power consumption of Bloom filters, we propose to employ pipelining technique in the architecture of Bloom filters. We call this new type of Bloom filters as *partitioned Bloom filters*. We provide a mathematical analysis to show that the partitioned Bloom filters is more efficient than the regular Bloom filters in terms of power. In this paper, we will first introduce Bloom filters, and then we propose a partitioned Bloom filter architecture. Following a power analysis of partitioned Bloom filters, a latency inquiry approach is given at the last section.

## II. Partitioned Bloom Filters

A Bloom filter is a data structure that stores a given set of signatures. It consists of a set of hash functions, a lookup array stored on SRAM, a hash buffer to store hash results temporarily, and a decision component made up of an AND gate. A Bloom filter is shown in Fig. 1.

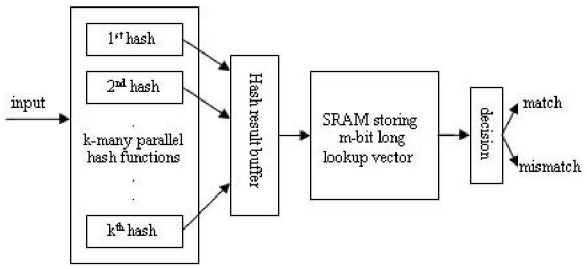


Fig. 1. A Bloom filter with k hash functions

Two operations are defined on a Bloom filter. First operation is called *programming* of the Bloom filter, and the second operation is *querying*. During the programming phase, a Bloom filter computes *k* many hash functions for each member of the signature set. Consequently, it uses these hash function results as an index into the lookup vector to set the bits up. In query phase, for each input a Bloom filter calculates *k* many hash functions, additionally checks the bits at the corresponding locations. If there is any one bit unset at the computed index positions, input is not member of the signature set. Otherwise all bits are found out to be set, it claims the input being a member of the set with a *false positive probability* [7].

Hash functions used in the Bloom filters are generally of type *universal hash functions* [4]. The performance of universal hash functions are explored by Ramakrishna et al [9]. The hash function, being a member of a universal hash function class, maps the input string to an output string, such that collision probability of given any two input strings is small. Given any string *X*, consisting of *b* bits,

$$X =< x_1, x_2, x_3 ... x_b > \quad (1)$$

$i^{th}$ hash function over the string *X* is defined as

$$h_i(x) = r_{i1} \bullet x_1 \oplus r_{i2} \bullet x_2 \oplus r_{i3} \bullet x_3 \oplus ... r_{ib} \bullet x_b \quad (2)$$

where $r_{ij}$'s are random coefficients ranging from *1* to *m*, and $x_i$'s are the bits in the input string.

A single Bloom filter uses *k* many hash functions in order to make a decision on the input. Hence the power consumption of a Bloom filter shown in Fig. 1 is a summation of the power consumptions of each of the hash functions, $P_{H_i}$, with the lookup operation including power dissipated at hash buffer, $P_L$, followed, plus an AND operation:

$$P_{BF_{regular}} = \sum_{i=1}^{k} (P_{H_i} + P_L) + P_{AND} \quad (3)$$

Power consumption of an AND gate is ignored hereafter, since it is minimal compared to the power used by the hash functions. We assume that the lookup power over a $m$-bit vector is approximately constant for each index calculated by any of the hash functions. The power consumption equation for a single Bloom filter simply becomes the total power used up by the hash functions and the power consumed by querying the $m$-bit vector for each hash value:

$$P_{BF_{regular}} = \sum_{i=1}^{k} (P_{H_i} + P_L) \quad (4)$$

The power consumptions of different hash function implementations in hardware have been measured by Yuksel [11]. In [11] author makes use of the multi-hashing [12] technique. Multi-hashing begins dividing the input into smaller pieces. Different universal hash functions from the same family applied to these input pieces. Eventually results are concatenated to form the desired hash value. Amongst the different hash functions analyzed in [11], the most power efficient hash function for a fixed frequency is *Weighted NH Polynomial with Reduction, or WH*. This hash function is a derivative of the universal hash functions.

In order to compare the power consumption of a regular Bloom filter to that of a partitioned Bloom filter, we use 16-bit implementation of hash functions. All of the *k* many hash functions are of type 16-bit WH hash functions, so Equ. 4 becomes

$$P_{BF_{regular}} = \sum_{i=1}^{k} (P_{H_{i(WH_{16})}} + P_L) = k.(P_{WH_{16}} + P_L) \quad (5)$$

## III. Power Analysis of Partitioned Bloom Filters

Since the number of hash functions required to minimize the false positive probability of a Bloom filter is large, it is better, in terms of power, to implement these hash functions in a pipelined manner. We call this new type of Bloom filters *partitioned Bloom filters*.

Basically, a partitioned Bloom filter, as shown in Fig. 2, consists of two groups of hash functions. The first stage always computes the hash values. By contrast, the second stage of hash functions only compute the hash values if in the first stage there is a match between the input and the signature sought.

The partitioned Bloom filters will have the same number of hash functions as the regular Bloom filters. Hence the false positive probability is the same. A partitioned Bloom filter exploits the virus free nature of the network traffic in most of the time. At worst, it will operate like a regular Bloom filter, which uses all of the hash functions before making a decision on the type of the input. However, most of the time the first group of hash functions will probably catch a mismatch between the input and the signature, claiming input is free of malicious content. Therefore the need to compute the second stage of hash functions diminishes. As a result, power consumption is reduced in a partitioned Bloom filter.

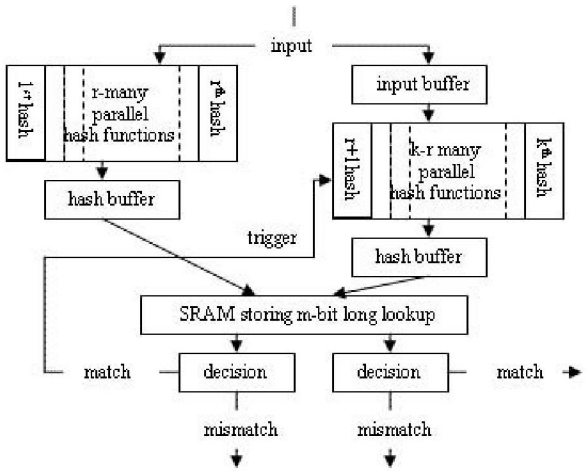


Fig. 2. A 2-stage partitioned Bloom filter

By following a similar analysis of [8], we begin to derive the total power consumption of the partitioned Bloom filter. We assume that the hash functions used in each Bloom filter is perfectly random. This is a reasonable assumption since each hash function coefficients are selected randomly in range 1 to *m*, where *m* is the number of bits in lookup vector. Overall, we have *k*-many random universal hash functions in a partitioned Bloom filter. In the first stage, *r*-many of them are utilized. The number of signatures sought in a Bloom filter is given as *n* as before.

Let us first derive the probability of match in the first stage. The probability that a bit is still unset after all the signatures are programmed into the partitioned Bloom filter by using *k*-many independent hash functions is $\alpha$.

$$\alpha = \left(1 - \frac{1}{m}\right)^{kn} \approx e^{\frac{-kn}{m}} (for\ large\ m) \quad (6)$$

where $\frac{1}{m}$ represents any one of the *m* bits set by a single hash function operating on a single signature. Then $\left(1 - \frac{1}{m}\right)$ is the probability that the bit is unset after a single hash value computation with a single signature. For it to remain unset, it should not be set by any of the *k*-many hash functions each operating on all of the *n*-many signatures in the signature set. Consequently, the probability that any one of the bits is set is

$$(1 - \alpha) \approx 1 - e^{\frac{-kn}{m}} \quad (7)$$

In order for the first stage to produce a match, the bits indexed by all *r* of the independent random hash functions

should be set. So the match probability of the first stage is, represented as $p$,

$$p = \prod_{i=1}^{r}(1-\alpha) = (1-\alpha)^r \approx (1-e^{\frac{-kn}{m}})^r \quad (8)$$

The mismatch probability of the first stage is simply *1-p*,

$$1-(1-e^{\frac{-kn}{m}})^r \quad (9)$$

With a probability of *(1-p)* the first stage of the hash functions in the partitioned Bloom filter will produce a mismatch. Otherwise, the first stage produces a match, then the second stage is used to compare the input with the signature sought. Therefore the power consumption of a partitioned Bloom filter is given by

$$\begin{aligned} P_{BFpartitioned} &= P_{1st-stage} + P\{match\} \times P_{2nd-stage} \\ P_{BFpartitioned} &= \sum_{i=1}^{r}(P_{H_i}+P_L) + p \times \sum_{j=r+1}^{k}(P_{H_j}+P_L) \\ &\quad + P_{AND} \end{aligned} \quad (10)$$

Again, $P_{AND}$ is omitted, since it is small with respect to the power consumption of hash functions. The power consumption of a single partitioned Bloom filter simplifies to

$$P_{BFpartitioned} = \sum_{i=1}^{r}(P_{H_i}+P_L) + p \times \sum_{j=r+1}^{k}(P_{H_j}+P_L) \quad (11)$$

Again for comparison purposes, $P_{H_{i,j}}$ are of type 16-bit WH. By substituting Equ. 8 into Equ. 11 power consumption of a partitioned Bloom filter becomes

$$\begin{aligned} P_{BF_{partitioned}} &= \sum_{i=1}^{r}(P_{H_{i(WH_{16})}}+P_L) \\ &\quad + (1-e^{\frac{-kn}{m}})^r \times \sum_{j=r+1}^{k}(P_{H_{j(WH_{16})}}+P_L) \\ &= r.(P_{WH_{16}}+P_L) + \\ &\quad (1-e^{\frac{-kn}{m}})^r(k-r)(P_{WH_{16}}+P_L) \end{aligned} \quad (12)$$

The power saving ratio, *PSR*, in a single Bloom filter deploying pipelining technique can be calculated as

$$PSR = \frac{(P_{regular} - P_{partitioned})}{P_{regular}} \quad (13)$$

By substituting Equ. 5 and Equ. 12 into Equ. 13, the average power saving ratio, *PSR*, is given by

$$PSR = \frac{\left(k \times A - \left[r + (1-e^{\frac{-kn}{m}})^r \times (k-r)\right] \times A\right)}{k \times A} \quad (14)$$

where $A = (P_{WH_{16}} + P_L)$, which is the power consumption of a single hash function with a single lookup operation. After simplifying *A*s, average power saving ratio, *PSR*, is found out to be

$$PSR = \frac{k - r + (r-k)(1-e^{\frac{-kn}{m}})^r}{k}$$

For different values of the number of bits allocated to per signature, $\frac{m}{n}$, power savings over the number of hash functions utilized in the first stage are illustrated in Fig. 3.

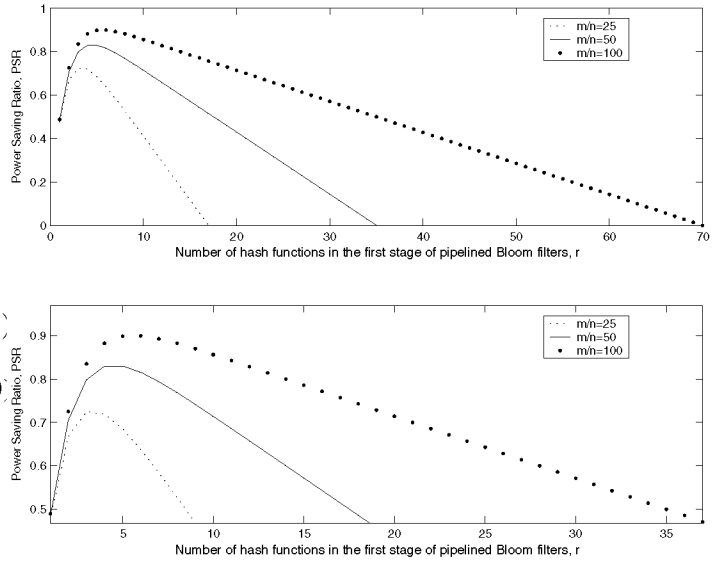


Fig. 3. Power saving ratio (PSR) in partitioned Bloom filters w.r.t. number of hash functions in the first stage: actual and zoomed PSR

As it is seen in the Fig. 3, the number of bits per signature, $\frac{m}{n}$, increases, the amount of power conserved in the system increases. In other words, the power saving ratio becomes larger as $\frac{m}{n}$ increases. This is because, as $\frac{m}{n}$ increases, although probability of mismatch in first stage stays the same for all configurations for a fixed value of $r$ (See Equ. 9), the number of hash functions deployed in the first stage becomes a smaller portion of the overall hash functions deployed in each configuration. For a fixed value of $r$, $\frac{r}{k}$ decreases. This explains the reduction in power consumption. The less are the number of hash functions deployed in first stage compared to the overall system, the more the power is saved.

Another fact illustrated in Fig. 3 is that as the number of hash functions utilized in the first stage increases, the power saving ratio, *PSR*, first increases to an optimum *PSR* value, thereafter it drops gradually. The increase in the power saving ratio to an threshold value stems from the fact that increasing the number of hash functions in the first stage increases the probability of mismatch, thus the second stage is not utilized to decide on an input. After this optimum is exceeded, *PSR* decreases steadily. This is again because, the more hash functions are deployed in the first stage, the more power that they consume. If we increase the number of hash functions used in the first stage to such a degree that all hash functions in the system deployed in the first stage, there remain

no power gain at all (i.e., the system behaves just like a regular Bloom filter.)

## IV. Latency Analysis of Partitioned Bloom Filters

Generally, latency of a pipelined architecture depends upon the number of the stages in the architecture. However, when it comes to the average latency analysis, not only the number of the stages but also the frequency that the stages are utilized becomes a crucial factor. As a result, average latency is formulated as,

$$L_{average} = \sum_{i=1}^{N} L_i \times f_i \tag{17}$$

In Equ. 17, $N$ represents the number of the stages in the architecture, $L_i$ shows the latency of architecture with that many stages, $f_i$ appears as the stage utilization frequency. Since the proposed partitioned Bloom filter architecture in this paper has two stages, $N$ is two. Whenever there is a mismatch in the first stage, the second stage is never used. In other words, the second stage is only used when there is a match in the first stage. This occurs almost with a frequency equal to the rate of input being malicious due to a very small false positive rate on the Bloom filter. Assuming each stage takes exactly 1 clock cycle, average latency of the proposed architecture is given by

$$L_{average} = 1 \times (1 - \rho) + 2 \times \rho \tag{18}$$

where $\rho$ is the rate that input is malicious. For a real network, $\rho$ is really small. Apparently, latency of the partitioned Bloom filter is really close to that of a Bloom filter with a single stage. Table I shows some latency values for different values of $\rho$.

TABLE I
Latency vs $\rho$, Input Maliciousness Rate

| $\rho$ | latency (clock cycles) |
|---|---|
| 0.01 | 1.01 |
| 0.05 | 1.05 |
| 0.1 | 1.1 |

## V. Conclusion

In this paper, we have proposed a technique to pipeline the hash functions in the Bloom filters. Analytical results show that the pipelining technique significantly decreases the total power consumption of a Bloom filter. The parameters like the bits allocated to per signature and the number of hash functions have been shown to affect the power saving ratio drastically. In fact, the lesser the number of hash functions in the first stage, the more the power saving is. Consequently, the power saving ratio increases. Power analysis revealed that power savings can reach up to 90%. Latency analysis of the architecture has shown that the latency of the overall system does not increase owing to the nonmalicious input characteristic. The area of the partitioned Bloom filter is almost the same as that of the Bloom filter except for the buffer. Selection of the hash functions to be included in the first stage is not covered in the current analysis, and left as a future work.

## References


[1] M. Attig, S. Dharmapurikar, and J.L. Lockwood. "Implementation Results of Bloom Filters for String Matching", *Proc. of IEEE Symposium on Field-Programmable Custom Computing Machines*, pp. 322-323. Washington, DC., 2004.

[2] B. Bloom, "Space/Time Trade-Offs in Hash Coding with Allowable Errors", *Commun. ACM*, vol. 13, no. 7, pp. 422-426, July 1970.

[3] A. Broder and M. Mitzenmacher, "Network Applications of Bloom Filters: A Survey", *Internet Mathematics*, vol. 1, no. 4, pp. 485-509, July 2003.

[4] J. L. Carter and M. Wegman, "Universal classes of hash functions", *Journal of Computer and System Sciences*, vol. 18, pp. 143-154, 1978.

[5] S. Czerwinski, B. Y. Zhao, T. Hodes, A. D. Joseph, and R. Katz. "An Architecture for a Secure Service Discovery Service", *Proc. ACM/IEEE International Conference on Mobile Computing and Networking Transactions on Networking* , pp. 24-35. New York, 1999.

[6] L. Fan, P. Cao, J. Almeida, A. Z. Broder. "Summary Cache: A Scalable Wide-Area Web Cache Sharing Protocol", *IEEE/ACM Transactions on Networking*, vol. 8, no. 3, pp. 281-293, 2000.

[7] S. Dharmapurikar, P. Krishnamurthy, T.S. Sproull, and J. W. Lockwood, "Deep Packet Inspection Using Parallel Bloom Filters", *IEEE Micro*, vol. 24, no. 1, pp. 52-61, 2004.

[8] M. Mitzenmacher, "Compressed Bloom filters", *IEEE/ACM Transactions on Networking*, vol. 10, no. 5, pp. 604-612, October, 2002.

[9] M. Ramakrishna, E. Fu, and E. Bahcekapili, "Efficient Hardware Hashing Functions for High Performance Computers", *IEEE Transactions on Computers*, vol. 48, no. 12, pp. 1378-1381, 1997.

[10] The Sourcefire Vulnerability Research Team, "Official Snort Ruleset", Sourcefire, Inc., Columbia, MD, August 2005. (web: http://www.snort.org/pub-bin/downloads.cgi)

[11] Kaan Yuksel, "Universal Hashing for Ultra-Low-Power Cryptographic Hardware Applications", *MS Thesis*,Worcester Polytechnic Institute, 2004.

[12] P. Rogaway, "Bucket Hashing and Its Application to Fast Message Authentication", *Journal of Cryptology*, vol. 12, no. 2, pp. 91-116, 1999.